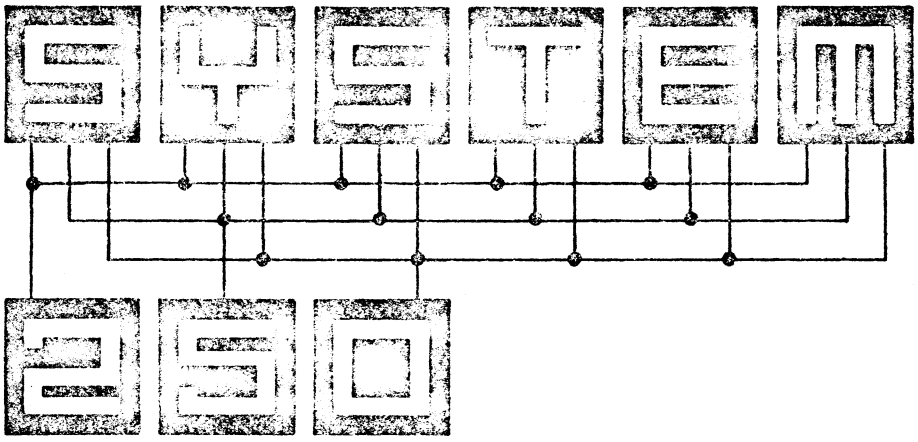

Architectural Features of System 250

D. M. England - System Design Manager

(Published in the Infotech State of the Art Report on Operating Systems, 1972)

Plessey Telecommunications Research Ltd.,
Taplow Court, Taplow,
Maidenhead,
Berks.

ARCHITECTURAL FEATURES OF SYSTEM 250

Abstract: The paper describes the hardware and software modules of System 250, a multi-processor, multi-user system designed for reliable real-time operation.

D. M. ENGLAND - System Design Manager.
Plessey Telecommunications Research - Taplow Court.

INTRODUCTION

1. System 250 is a major processing system development. It is part of the stored program control telephone switching programme that the Plessey Company has undertaken in cooperation with N.R.D.C. As well as telephone switching, the system has wide application in the real-time sector, particularly where there are requirements for fault tolerance, expansion without discontinuity or reprogramming, and the ability to handle a large number of devices of varying characteristics.

2. From the outset, System 250 has been designed as a complex of hardware and software. Careful attention has been paid to the inclusion of hardware mechanisms which support and simplify software operation. The operating system is then a body of software which exploits the hardware to present a logical and convenient set of facilities to the user.

3. This paper sets out to identify the architectural and system design features of System 250 and to describe the normal operation of its constituent hardware modules and software packages. No more than passing reference is made to the question of system security and high availability, as this is outside the scope of the present paper.

SYSTEM HARDWARE CONCEPT

4. System 250 is a multi-processor. It may be configured with a number of processors, stores, etc., according to the traffic characteristics of a given installation. The number of hardware modules in an installation may be increased to match traffic growth. Yet all installations share a common architecture, including a common operating system. The system concept, as it is expressed in hardware, is described by reference to a typical configuration of modules (Fig.1):-

CPU Module

5. The system includes one or more identical 24 bit word "CPU modules". Normally there are at least two for redundancy. Principal system design features are as follows:-

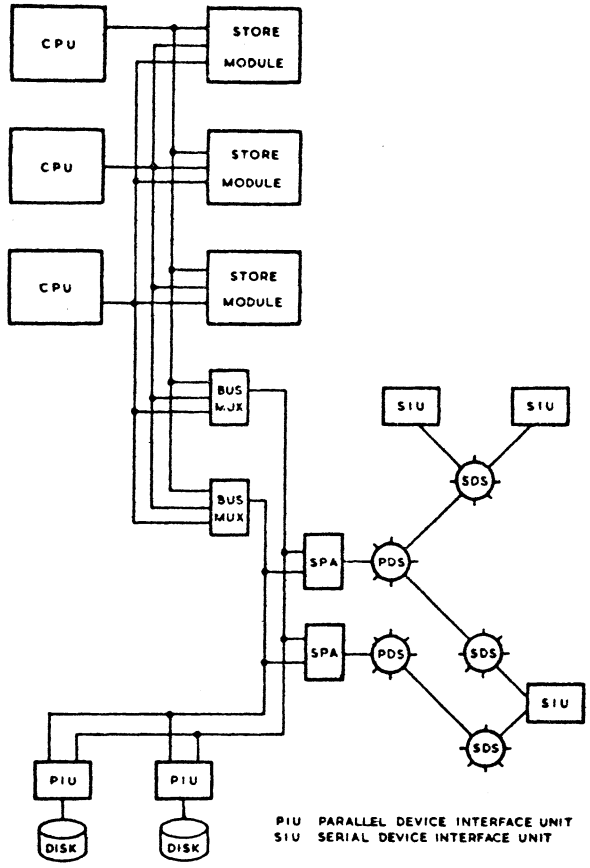

FIG 1. TYPICAL HARDWARE CONFIGURATION

(1) A CPU can communicate with any store module by means of its own parallel "CPU bus".

(2) The CPU module includes an implementation of base registers for addressing store that permits all store to be dynamic and all code reentrant. These are called "capability registers".

(3) Multiple CPUs share the system workload from a common work pool in store. All CPUs are equal and no function is ever dedicated to a particular CPU.

Store Module

6. The system includes one or more "store modules" of up to 64K each. Normally there are at least two, for redundancy. Each store is interfaced to all CPU buses by an "access unit". This contains address recognition logic, so that it can detect and service access demands on that store. It is also able to queue demands where CPUs contend for access to that store.

Bus Multiplexor

7. The "bus multiplexor" is an access unit giving access from CPU buses to the bus multiplexor's parallel "peripheral bus" and thence to devices interfaced to the peripheral bus. To preserve access to devices in the event of failure, bus multiplexors and peripheral buses are normally duplicated.

Parallel Devices

8. Each device is interfaced to both peripheral buses by an access unit. Thus, each device looks to a CPU like a store module containing a small number of words, which are in fact the command, status, and data registers of the device controller. The system contains no IO instructions; CPU instructions like load data and store data are sufficient to communicate with these registers, the only difference from a store module being the curious effects of such communication.

Device Polling

9. An input (or output) transfer on a particular device, e.g. a disk, may be achieved by a "polling program" running on one of the CPUs, polling a full/empty binary in the corresponding status register, and transferring the next word or character when it indicates full (or empty).

Channelling

10. Being a multi-processor, the system requires no autonomous channels or IO processors; it employs CPU power to perform the channelling function. The CPU module is enabled to perform simultaneous transfers on several devices by provision of two hardware mechanisms:-

(1) A first-in-first-out buffer store in each device, that is filled from (or emptied into) the device at its own speed. This buffer is emptied into (or filled from) main store by the polling program.

(2) The ability of the block transfer instruction in the CPU to detect the empty (or full) state of the buffer store as a terminal condition of the instruction. The block transfer may then be used within the polling program to drain or top up a number of device buffers in rotation. This introduces the concept of a "channelling program" in place of autonomous channels.

Serial Medium

11. To provide the ability to interface a large number of low activity devices (e.g. teletypewriters, relay banks), the system features a serial data collection and distribution medium culminating in a "serial to parallel adapter" (SPA), which is itself a device interfaced to the peripheral buses:-

(1) This "serial medium" consists of a network of primary (64-way) and secondary (16-way) "data switches" (PDS and SDS) to which serial devices can be interfaced.

(2) The appearance of input data is detected at a serial device interface and automatically transferred on the serial medium to the SPA.

(3) Data is in the form of variable length (e.g. word or character) data messages, which are collected in or distributed from an input or output buffer respectively in the SPA. In order to know its source or destination, each data message has attached to it the identity of the device concerned, comprised of the addresses of the primary and secondary switch ports that provide its switch path through the serial medium.

(4) As a device on the parallel buses, the SPA has its input buffer emptied and its output buffer filled at suitable intervals by a polling program.

(5) To preserve access to serial devices in the event of failure, normally an alternative path to each device is provided by duplication of the SPA and serial medium.

Interrupts

12. System 250 has no external or device interrupts in the conventional sense:-

(1) The serial medium has the effect of servicing interrupts at the periphery and recording them in the SPA for processing at the time allotted by the system.

(2) In the absence of autonomous channels there is no requirement for channel interrupts, e.g. disk

interrupts. On completion of a transfer, the channelling process is immediately capable of initiating a further transfer on the same device.

Interval Timer

13. The requirement of any processing system to measure time intervals is met in System 250 by an "interval timer" resident in each CPU. An interval timer's register may be loaded with a time value that is automatically decremented in units of 100 μsecs. When it reaches zero the program running on that CPU is interrupted and the timeout is processed. It should be noted that although the channelling program runs interrupt inhibited, provision is made for the servicing of any timer interrupt at convenient points.

CAPABILITY CONCEPT

14. The root problem in the design of a multi-processor is to find a method of sharing store between CPUs which limits the scope for store corruption possessed by a faulty CPU without limiting the freedom of the software designer to structure his data and code in the most natural and logical manner.

15. In System 250 the solution to this problem is the "capability". The capability concept is the most important feature of the system design, over the whole of which it has exercised a considerable influence. It is at the heart of many solutions to system design problems. Far from limiting freedom it enhances the power of the software designer to create explicit and coherent program structures.

Capability Registers

16. As well as eight conventional "data registers", there are in the CPU module eight "capability registers". The purpose of a capability register is to provide a means of addressing store. The format of a capability register is shown in Fig. 2:-

(1) A capability register defines the lower and upper bounds, base and limit, of a block wholly contained within a store module (Fig.2a). It also defines, in its "access field", the operations which can be performed on that block.

(2) The words within the block can now be referred to in instructions; the address construction takes the base value from the specified capability register, and using it an an addressing base adds the address offset and (optionally) a modifier value to give the absolute address of the word referred to (Fig. 2b).

(3) The access field is a six bit code, whose constituent bits define the operations that are permitted on the block concerned, in any combination (Fig. 2c). The "read data" and "write data" access types permit operations between the store blocks and the data registers, "load data" and "store data" being the most obvious. Similarly, "read capability" and "write capability" permit load and store operations between the store block and capability registers. The "execute" type permits the block to be executed as program code. The "enter" type needs some elaboration and is discussed in para. 24 below.

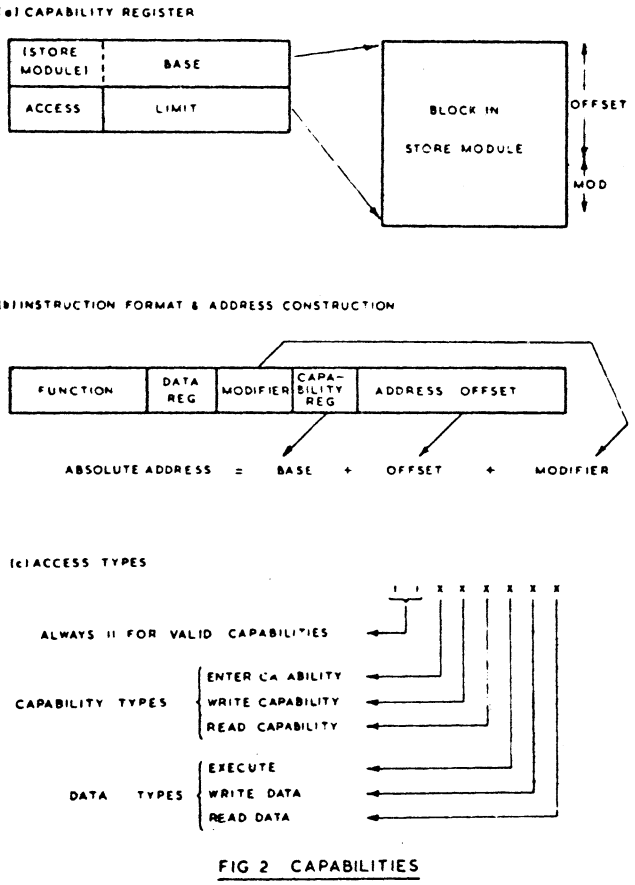

FIG 2 CAPABILITIES

17. Thus, a capability register defines the permitted bounds of a store block and the operations permitted upon it. Any attempt by a program to refer outside these bounds or to perform illicit operations is detected by hardware in the CPU and the program is interrupted into a fault handler. The capability register can now been seen to have two functions:-

(1) To provide an addressing base for access to the store, as discussed above.

(2) To limit the scope of a program and thus contain its potential for store corruption in the event of an error.

18. It should be noted that if an access field permits data to be written into a block and read out again as a capability value, then it is possible to manufacture capabilities and thence gain access anywhere in the storage system. Such access fields are consequently forbidden; a block is very strictly either a capability type block or a data type block.

19. A capability, then, is an access right to a store block. It can be held in a capability block, just as data is held in a data block. It can be "loaded" into a capability register when the block it defines requires to be accessed. The operation of a CPU's load capability instruction is depicted in Fig. 3, described as follows.

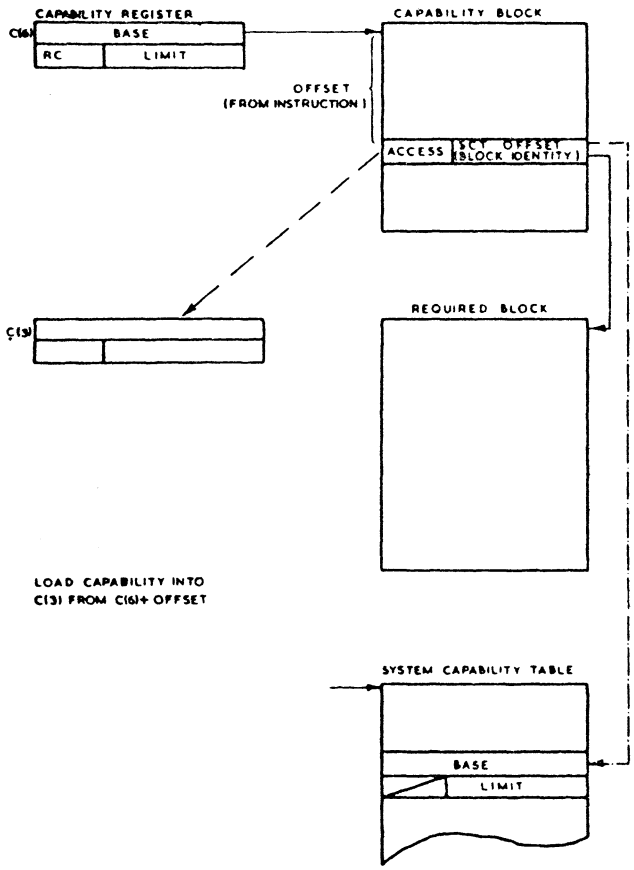

FIG 3 LOAD CAPABILITY

(1) The base and limit values of all blocks within the main storage system are contained within a special block called the "system capability table", which is defined by a special capability register.

(2) The required capability, held within a capability block, instead of containing the actual base/limit values, contains an offset reference to the system capability table entry containing the base/limit values. The capability does however contain its own access type for the required block.

(3) The effect of the load capability instruction is to take the access field from the capability and the base/limit values from the system capability table and put them together in the required capability register.

Capability Structure

20. The concept of a capability as an access right to a store block may now be dissociated from the physical location of the block in the storage system. The system capability table is accessed automatically by the load capability instruction and the programmer need know nothing about it. He can imagine a capability register to have the same format as a stored capability, consisting of an access field and a block identity field (Fig. 4). Conceptually, a capability (access right) is loaded into a register when access to the corresponding block is required. Equally, a capability held in a register can be stored in a capability block until access is required to the block defined.

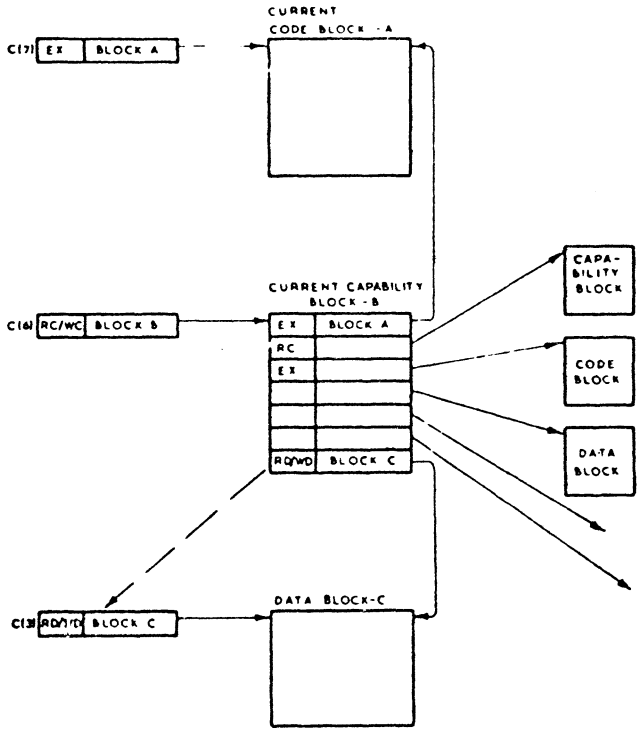

FIG 4 CAPABILITY STRUCTURE

21. By hardware conventions, one of the eight capability registers (C7) defines the code block currently being executed and another defines a current capability block (A and B in Fig. 4), containing capabilities for blocks that the code may want to operate on. Some of these would be data blocks or other code blocks that might have to be executed. Some might themselves be capability blocks,giving access rights to a whole new range of blocks. Following through the idea that a capability for one capability block can provide access to further capability blocks and so on, a whole network can be constructed consisting of code and data blocks interconnected by capability blocks, as shown in Fig. 5.

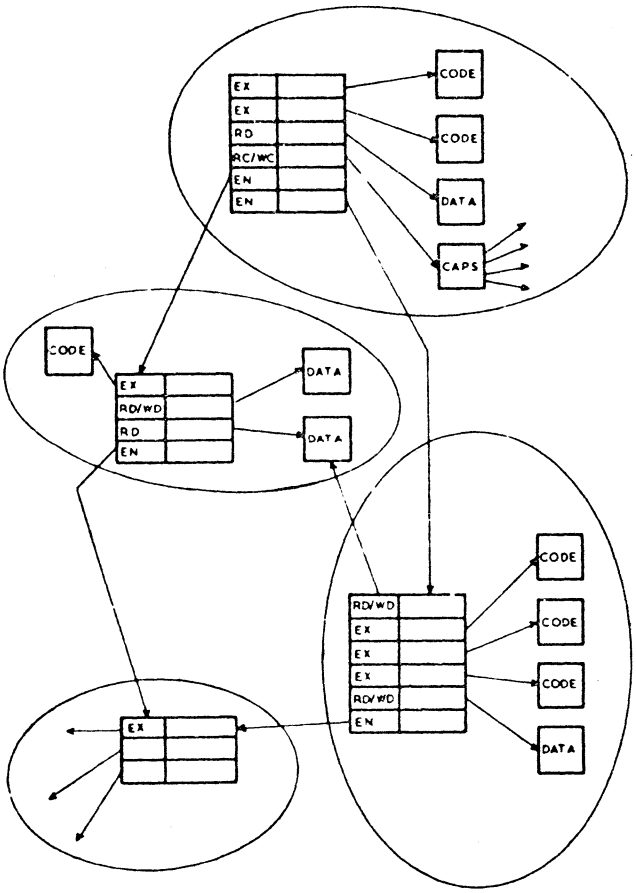

FIG 5 CAPABILITY NETWORK

22. Looking more closely at this network, each of its nodes consists of a capability block with satellite code and data blocks, and even satellite capability blocks giving access to further data blocks, etc., the whole forming a nodal data structure on which its code blocks can operate. A code block can operate on this data structure so long as two capabilities are supplied:-

(1) For the code block itself, in C7.

(2) For the node's main capability block, in C6, giving access to all blocks of the node.

23. To see how these capabilities are supplied the "enter" mechanism must be considered.

Enter Mechanisms and Dump Stack

24. One of the available access types for a store block is "enter" (Fig. 2c). Its purpose is to permit a subroutine call from one node to another, or more strictly a code block in one node to a code block in another, as follows:-

(1) The calling node has an enter type capability for the main capability block of the called node (See Fig. 5). An offset down this block gives an execute type capability for the required code block.

(2) The call is achieved by a CPU instruction "call", specifying the enter type capability and offset. The effect of the call instruction is to load C7 with the execute type capability for the required code block and to load C6 with the enter type capability for the called node's main capability block. A read capability access type is automatically supplied in C6, so that the called node can read its own capabilities.

(3) The old values of C6 and C7, which define the current capability and code blocks of the calling node prior to the call, are automatically preserved in a stack by the call instruction. These values are automatically restored when the called node performs the CPU instruction "return".

25. It should be noted that possession on an enter type capability does not permit the reading of capabilities from the block defined; a calling node cannot rifle a called node's capabilities and gain access to its blocks. Further, the two nodes are mutually protected from one another; they are denied access to one another's blocks by both "call" and "return" overwriting C6 and C7. The remaining capability registers (C0-C5) can however be used to carry parameters across the interface between nodes.

26. The stack employed to preserve C6 and C7 values is unique to the execution of a program, during which the block containing it is defined by a special capability register. This block is called a "dump stack", and besides the stack is used to preserve register contents on interrupt or context change.

Capabilities for Parallel Devices

27. A peripheral device interfaced to the peripheral buses has the appearance of a small store module (para. 8). Consequently, like any other block in the storage system, its register set is defined by a capability, permitting it to be operated on by standard CPU instructions and protecting it against unauthorised access.

OPERATING SYSTEM STRUCTURE

28. The operating system is designed as a modular set of reentrant program packages which both run on and control a hardware configuration as described above. The broad structure of the operating system is depicted in Fig. 6 and is in two parts:-

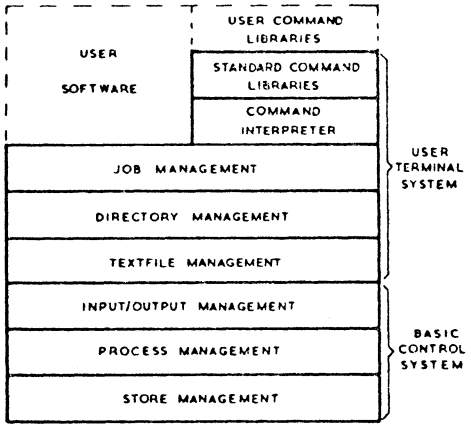

FIG 6 BROAD STRUCTURE OF OPERATING SYSTEM

(1) Basic Control System - The purpose of the "basic control system" is to provide the user with convenient standard facilities for management of storage, processing power, and input/output resources without him being concerned with the mechanics of controlling hardware modules. Moreover, the operating system renders a user program unconscious of the hardware modules which furnish its resource requirement, and therefore insensitive to any changes in the number of CPUs and storage modules which are on-line at any given time.

(2) User Terminal System - The purpose of the "user terminal system" is to provide a standard, recursive syntax in which the user can express commands to System 250 (Fig. 7 shows examples of commands). This is achieved by means of a standard "command interpreter" package, which interprets each command and obeys it by executing an associated "command program". The operating system includes a set of commands which enable the user to define his own command programs, resulting in a "command language" that is freely extendable according to the user's command requirements. Certain other standard "command libraries" are available within the operating system to provide program development and job control facilities. Because it is fully reentrant, the operating system can support an arbitrary number of active terminals, thereby achieving a multi-access utility.

```
LOGIN SMITH JOB15
ASM BINSEARCH (
.. BINARY SEARCH SUBROUTINE
INPUT:  C1 CONTAINS CAPABILITY FOR BLOCK
        D2 CONTAINS PATTERN TO BE MATCHED
        D3 CONTAINS ADDRESS OF MIDDLE ELEMENT
        INDICATORS:  ZERO SET IF MATCH, NONZERO SET
                                IF NO MATCH..

        LD    D1   0            ..SET START ADDRESS..
LOOP:   OR    D1   D3           ..OR IN TRIAL INCREMENT..
        CMP   D2   TBL  D1  C1  ..COMPARE WITH ELEMENT..
        JEQ   FINISH            ..JUMP OUT IF MATCH..
        JGT   UPPER             ..JUMP IF UPPER HALF..
        EOR   D1   D3           ..LOWER HALF - REMOVE INCREMENT..
UPPER:  LSH   D3   -1           ..SHIFT INCREMENT RIGHT ONE PLACE..
        JNE   LOOP              ..REPEAT IF NOT FINISHED..
        CMP   D2   TBL  D1  C1  ..SET INDICATORS..
FINISH: RET                     ..RETURN TO CALLER..)

PRINT OLDFILE

BLOCK FRED 615 RWD

PROCESS ANALYSER CAPX CODEX

AT JOE 3 (PRINT FRED (2*P)
          N:=(N-1)
          If N<0 THEN (REMOVE JOE 3))

LOGOUT
```

FIG. 7. EXAMPLES OF COMMANDS

RESOURCE MANAGEMENT

29. The principal system design features of operating system packages are described below:-

Store Management

30. (1) Dynamic Storage - The "store management package" employs the capability mechanism to achieve a dynamic storage scheme in which a program may request unique blocks of store at run time and be delivered capabilities for them giving the required access rights. It should be remembered that the capability mechanism protects this dynamic data structure from unauthorised access by other programs.

(2) Virtual Store - The capability concept is extended from main store to disks, permitting the capability structure to extend over the whole of the storage system. This achieves a virtual storage scheme, in which the distinction between main and backing stores is transparent to user programs. Initiating transfers between the two is the responsibility of the store management package. A block is moved in when an attempt is made to access it or moved out again to maintain a reservoir of directly accessible storage. The blocks selected for moving out are those that have remained longest in main store without being accessed.

(3) Capabilities on Disk - The format of a capability on a disk includes disk identity and address fields in place of the system capability table offset (para. 19). Whenever a capability block is moved in or out its constituent capabilities are transformed accordingly. This means that a capability in main store must have a corresponding system capability table entry, even though the corresponding block may only exist on disk. An attempt to access such a block is detected by a simple hardware mechanism, and the block is moved in to permit the accessing program to proceed.

(4) Deallocation and Garbage Collection - When a program requests a store block, the store management package allocates disk space for it. When subsequently the program explicitly releases the block, this disk space is rendered reusable by invalidating all existing capabilities that refer to it (This is achieved by a disk sector sequence number mechanism). A background "garbage collection program" seeks out and destroys invalid capabilities and blocks that have been severed by program error from the main capability network of the system.

Process Management

31. (1) Processes - The dynamic allocation of blocks containing the active data of a running program permits that program to be reentrant. The term applied to the execution of a reentrant program is a "process", and for a given program there may be many processes simultaneously in existence at any instant. The "process management package" is concerned with the creation, scheduling, and synchronisation of processes.

(2) Scheduling - It should be noted that the common work list (para. 5) employed to achieve a load sharing philosophy for CPUs is actually a priority structured "process ready list" of processes in an executable condition. It is transparent to the process at the head of the list which CPU happens to schedule and execute it. Thus, the introduction of a further CPU simply increases the available processing capacity by increasing the number of processes that can be executed simultaneously. It has no effect on the structure of System 250 programs.

(3) Synchronising Flags - The facility provided by the process management package for communication between processes is a "flag"; one process "posts" a message "on a flag" and the other "waits for" it. Synchronisation for message passing between asynchronously operating processes is achieved by permitting "posts" and "wait for" to occur in arbitrary order; a flag may then hold a message until "waited for" or hold a process in suspension until the message is "posted". In fact, a flag may hold a number of separately posted messages and may be accessible by several processes, so that a flag may be viewed as a data structure consisting of either a queue of posted messages, that are to be waited for in turn, or a queue of suspended processes waiting for messages to be posted.

(4) Time Intervals and Multiple Events - The event awaited by a process may be expiry of a time interval rather than delivery of a message. In this case, the process is held in a time ordered chain of suspended processes. This "time chain" is serviced by "interrupt processes" in response to interval timer interrupts (para. 13). A process is permitted to wait for one of a number of possible events, e.g. a telephone switching process might wait for a message indicating the next dialled digit, or a message that the subscriber's handset had been returned to its rest, or expiry of a time interval indicating some abnormal condition. Fig. 8 depicts a multiple wait for situation; the suspended process (x) has entries in several queues, forming a circular list as shown. When one event occurs the others are automatically cancelled and the process is entered in the process ready list with a parameter set to indicate the event which occurred.

Input/Output Management

32. The "input/output management package" includes the polling/channelling programs discussed in para. 9ff and handlers for all parallel and serial devices:-

(1) Channelling - The transfer of data between main store and parallel devices (see para. 10) is effected by a "channelling process" (Fig. 9), which is able to support a number of simultaneous transfers according to an installation defined "channel schedule".

The "channels" are actually device handler subroutines and they are called in rotation, the action of a channel during a transfer being to drain or top up the associated device buffer. The channel also sets up a new transfer on completion of any previous transfer.

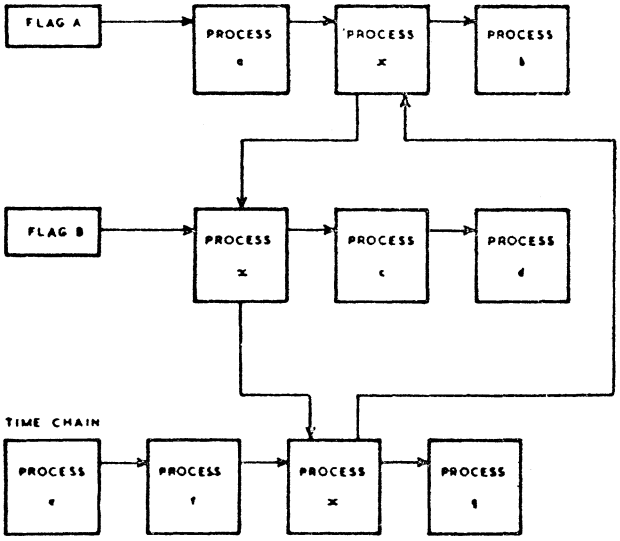

FIG 8 MULTIPLE WAIT FOR, A OR B OR TIME INCREMENT

(2) Disk Optimisation - A disk channel receives its transfer requests from the store management package; for example, there is a process which selects blocks for moving out. These requests are inserted in an optimisation queue according to the track/sector location of the block. On completion of a transfer, the disk channel is immediately able in software to select another transfer from the optimisation queue according to the sector position of the read/write heads. Thus, channelling by software achieves a high degree of optimisation without either the channel inefficiencies inherent in device interrupts or the alternative of complex channelling equipment.

(3) Channelling Processes and Channel Schedules Each channel must draw an amount of processing power dependent on the device speed. The total processing requirement of a channel schedule may not exceed the power of one CPU if the overflow of device buffers and consequent loss of data is to be avoided. Consequently, a system may include a number of channelling processes, handling separate sets of devices, which between them can occupy that number of CPUs. Moreover, optimisation of a channel schedule can be achieved by applying more than one channel to faster devices, i.e. calling their device handlers more than once per cycle. The important point to note is that, like hardware modules, channel schedules and processes are configurable for a given installation.

(4) Serial Medium Handling - A "serial medium handler" is a process which is applied at regular intervals to the data read from an SPA input buffer (para. 11). The structure of a serial medium handler is an exact mapping of the associated serial medium, consisting of a cascade of data switch handlers terminating in serial device handlers (Fig. 10). Each handler is a node of the capability network, and each data switch handler includes a "switch list" of enter type capabilities corresponding to the ports of the data switch. For each data message, using the port addresses given in the device identity as switch indexes, the serial medium handler switches to the associated device handler to process the message.

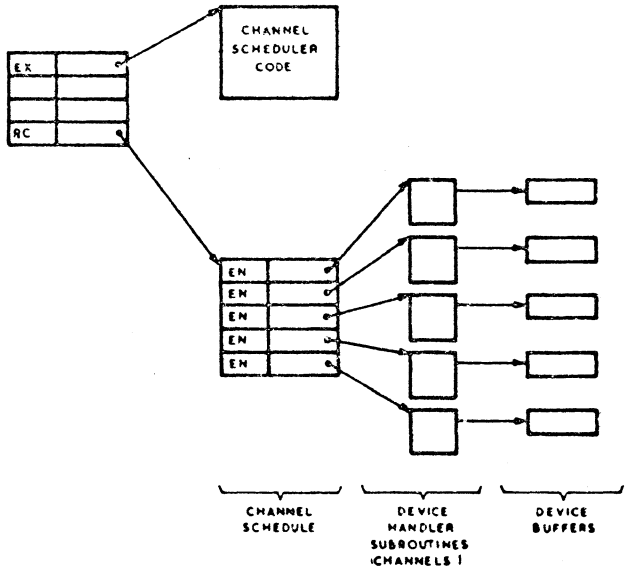

FIG 9 SOFTWARE CHANNELLING

(5) Serial Medium Status Messages - The first two data bits (say) of a message may be used to identify different registers in the device, e.g. data and status registers. This is particularly important for certain output devices, e.g. paper tape punch, as the input of a status message is used to indicate to the device handler that the device is ready for the next output message. The device handler forms the next output message, ready for transfer to the SPA output buffer. Thus, the serial medium handler handles both input and output messages, and output is made to look almost identical to input.

(6) Serial Medium Handlers and Switch Lists - A system may include a serial medium handler process for each active serial medium. Furthermore, each serial medium handler's switch lists are configurable

for a given installation in a manner analogous to the serial medium hardware modules.

(7) Spooling - A paper tape reader or punch or lineprinter is normally allocated to a "spooling process". The set of such processes takes advantage of the virtual store to achieve a conventional spooling arrangement; user programs are thereby rendered device independent, and communicate their input and output through the medium of "data streams". It should be noted that a paper tape reader handler provides the communication medium between its serial medium handler and spooling process. Both processes have an enter type capability for the tape reader handler, and call different subroutines within it to buffer input characters and get lines of text respectively. Synchronisation of the two processes, to pass across lines of text,is achieved by means of a flag as described above.

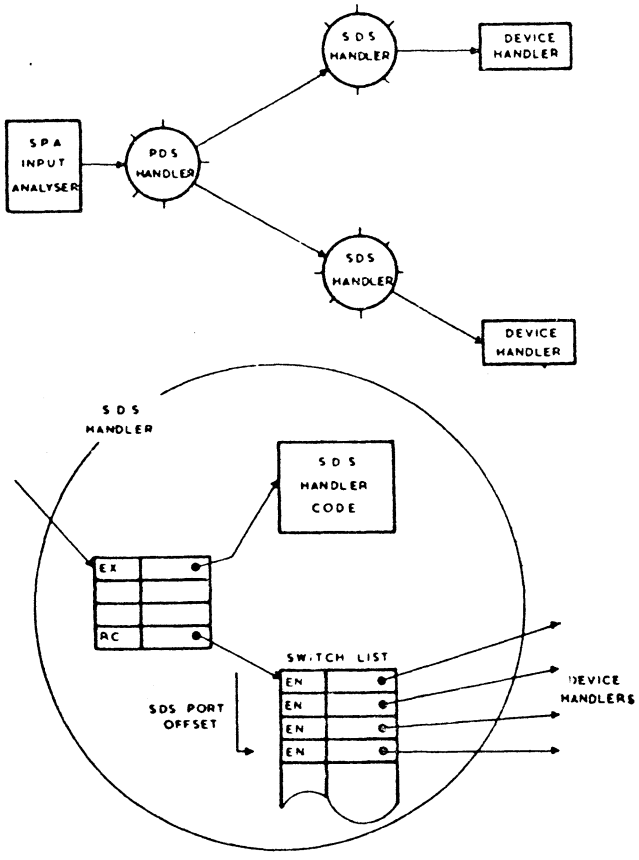

FIG 10 SERIAL MEDIUM HANDLER

File, Directory, and Job Management

33. The purpose of these packages is to provide facilities for the use of command programs, that are required to implement the user terminal system:-

(1) Job Management - The "job management package" is concerned with the management of a "job" or family of processes. All the processes constituting a job reference a common "job control block", which can be used to exercise control over all these processes at once, e.g. the setting of a particular bit stops any of them being scheduled and therefore suspends the job.

(2) Textfile Management - A "textfile" is a data structure containing lines of text characters, which may for example be source programs. The "textfile management package" provides facilities for creating and editing textfiles.

(3) Directory Management - The "directory management package" provides facilities for creating and maintaining "directories" of names referring to blocks, textfiles, jobs, other directories, etc. Directories are employed within the user terminal system to maintain a record of user defined names.

THE USER/RESOURCE INTERFACE

34. This section describes how the various resources provided by operating system packages are realised and represented and how they are accessed by user programs.

Resource Allocation

35. From physical system resources provided by a hardware configuration the operating system derives "logical resources" convenient to the user, namely:-

STORE BLOCK
PROCESS
FLAG
STREAM
TEXTFILE
DIRECTORY
JOB

The simplest of these resources is a single store block. Each of them consists of a data structure of blocks that is dynamically allocated by the corresponding operating system package when requested by a running process. Each package contains a "resource allocation subroutine" that may be called by means of an enter type capability

(para.24). A single read type capability gives a process access to all resource allocation subroutines via a "common facilities block" (Fig. 11).

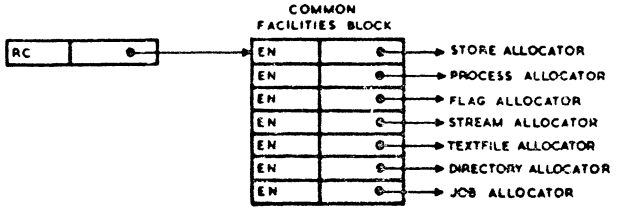

FIG 11 COMMON FACILITIES BLOCK

A Standard Software Interface

36. When a resource allocation subroutine is called within a running process it creates a suitable resource and returns a capability for that resource. In the case of a store block, this capability gives the access required to operate on words in the block in the normal way by CPU instructions. In the case of any other resource, the capability is of enter type and allows permitted operations on the resource by calling subroutines within the operating system package concerned (Fig. 12), e.g. to "post" and "wait for" messages on a flag. The following points are of particular note:-

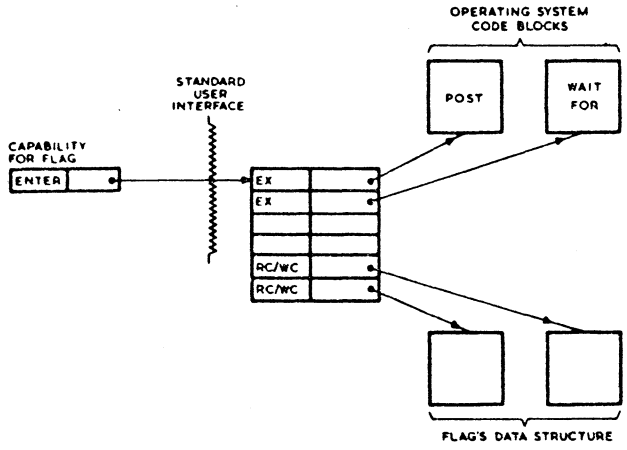

FIG 12 A RESOURCE & ITS INTERFACE, E.G. A FLAG

(1) System 250 has no privileged mode of operation; the protected nature of subroutine interfaces (para. 24) permits normal subroutine calls for entering operating system code blocks.

(2) The modular structure of operating system packages is explicit to the user; for each facility provided there is a corresponding code block.

(3) Every resource is represented by a single capability. Except for store blocks, every resource is a dynamically created node of the capability network, and the capability representing it provides a unique entry to the operating system for processing that resource alone.

(4) The capability provides a standard software interface, giving access rights to perform a permitted range of operations on the resource it represents, be it a single block or a complex data structure. Thus, in System 250 any resource (and the user can create his own) is accessed and manipulated in a standard manner regardless of the data structures and manipulations involved.

(5) It should be noted that to the user there is no distinction between manipulating store blocks by means of CPU instructions and other resources by means of operating system subroutines. The operating system simply provides a supplementary instruction set for manipulating more complex resources than a single block.

USER TERMINAL SYSTEM

Command Interpreter

37. When the user approaches a terminal and operates an "attention key", this causes a "command interpreter process" to be created, the purpose of which is to interpret and obey each command subsequently typed at the terminal keyboard. A command consists of a command name together with a string of parameters, any of which may themselves be commands (see Fig. 7). The program code of the command interpreter is recursive, so that it can preserve the state of one command to obey a nested command, to any level of nesting.

38. The command interpreter process requests a directory (para. 33(3)) to use as its "symbol table" (Fig. 13), in which to contain each symbolic name introduced at the terminal together with its associated capability or data value. For example, in Fig. 7 the parameter "LOOP" of the command ":" would have an associated label offset data value, and "FRED" would have an associated capability for a store block.

39. "Command names" are themselves recorded in the symbol table, with capabilities for associated "command programs" The action of the command

interpreter, in obeying each command typed at the terminal keyboard, is to execute the associated command program. A command program may either be a code block, which can be executed directly, or a string of commands that must be obeyed in turn.

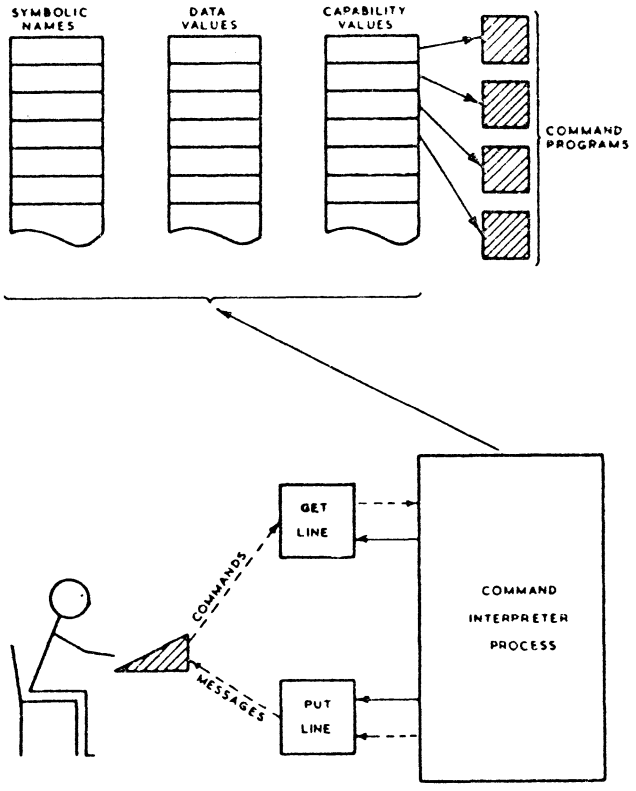

FIG 13 SYMBOL TABLE

40. When a command program is executed it calls for extraction and evaluation of the actual parameters from the command, at required points, by means of a "parameter evaluation" command. If the parameter is a symbol, its associated value is directly available in the symbol table. If it is a nested command, its command name is found in the symbol table and the associated command program is executed to deliver the required value.

Standard Command Libraries

41. The standard command libraries extend the facilities of operating system packages to the terminal, where these facilities are then available for the user to create and manipulate resources at the keyboard. By typing a command, the user gets the associated command program to call the operating system and employ its facilities on his behalf. The set of standard command libraries includes the following:-

(1) Assembly and Command Definition - The "assembly commands" are used to generate blocks of program code. There is an assembly command for each CPU instruction and each operating system facility. Obeying such a command generates the required binary instruction(s), and obeying a string of them within an "assemble command" (ASM in Fig. 7) causes program code to be assembled into a symbolically named store block (BINSEARCH in Fig. 7) allocated for the purpose. "Command definition" commands permit the user to create new commands compounded from existing ones. Besides allowing him to create his own command structures, these commands provide a conventional macro-assembly facility.

(2) Store and Process Commands - These are the analogue of the store management and process management facilities, and permit store blocks and processes to be created, symbolically named, and manipulated at a terminal keyboard.

(3) Directory Control - Were the user to "log out" from the terminal, the command interpreter and all its data structure, including the symbol table, would be released. To give permanence to his resources, the user is provided with a "directory" in which to preserve them. This is part of the system data structure. With this as the user's "base directory", he may employ "directory control" commands to create a structure of named directories (Fig. 14), from which he can retrieve his resources when required. The base directory is initialised to give access to system directories containing the standard command libraries. It may also be initialised to give access to directories that are common to a number of users, so that they can have resources in common, e.g. when writing different parts of the same software system.

(4) Job and Textfile Control - A symbol table may include capabilities for named processes, which can create further processes, etc., all of which constitute a job and reference its job control block (para. 33(1)). The job control block is created at the same time as the command interpreter process. The symbol table is referenced from the job control block, and is therefore part of the job's data structure. By giving the job a name, using a "job control" command, and preserving

it in a directory, the job is rendered independent of the command interpreter process; the user can "log out" from the terminal and leave the job running. When he "logs in" again he can restore control over the job by naming it; this furnishes the new command interpreter process with the original symbol table. Job control commands are available for naming the data streams (para. 32(7)) to be employed by command interpreter and user processes. Typically, a textfile (para. 33(2)) containing an ASM command might provide the input source to a command interpreter process. Commands are available to create and edit textfiles and also to insert monitor points and achieve selective print-outs for program testing purposes, e.g. the commands AT, PRINT, etc. in Fig. 7.

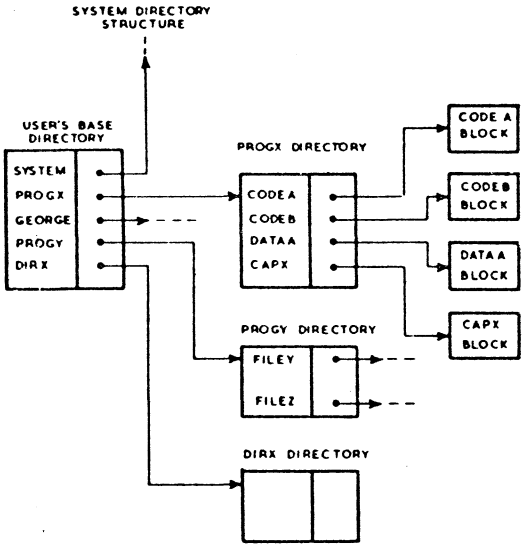

FIG 14 TYPICAL USER DIRECTORY STRUCTURE

Multiple Users and their Jobs

42. Finally, there are two points of particular importance regarding the nature of the user terminal system:-

(1) The whole of a user's data structure stems from his base directory and is protected by the capabilities that define it from unauthorised access by other users. This is equally true of a job's data structure, stemming from its job control block. Because the integrity of data structures is assured by their mutual exclusion, a given installation may support an arbitrary number of independent users, each of whom may have several discrete jobs. Nonetheless, the directory structure permits a conscious decision to share resources between users and/or jobs, with commensurate loss of independence.

(2) So long as a System 250 installation, possesses sufficient capacity to service all its commitments, each of the users' jobs may be a discrete real-time system, including dedicated serial and parallel devices (see para. 27). Device handlers for dedicated peripherals might be included within the self-contained data structure of each real-time job. Alternatively, the directory structure permits the sharing, for example, of a serial medium handler process and therefore the associated serial medium. This process might be configured (para. 32(6)) to include device handler subroutines belonging to several jobs.

ACKNOWLEDGEMENTS

43. The architectural simplicity and coherence of System 250 is due to the creative discontent of its designers, the wisdom and courage of their consultants, especially Professor M. V. Wilkes, Dr. R.S. Fabry, Mr. M. O'Halloran and Mr. M. Berry, and the patience and confidence of the Plessey Company.

BIBLIOGRAPHY

(1) J.M. Cotton — "The Operational Requirements for Future Communications Control Processors", Proceedings of International Switching Symposium, Boston, Mas., June 1972.

(2) D. Halton — "Hardware of the System 250" for Communication Control", ibid.

(3) D.M. England — "Operating System of System 250", ibid.

(4) W.A.C. Hemmings — "Telephone Switching Based on System 250", ibid.

(5) M.V. Wilkes — "Time Sharing Computer Systems", Macdonald.

(6) R.S. Fabry — "List Structured Addressing", Ph.D dissertation, University of Chicago, Illinois, June 1970.

(7) E. Dijkstra — "The Structure of 'THE' Multi-Programming System", Comm. ACM Vol, 11, No. 5, May 1968, pp 341-346.